Chapter SEVEN
Collections

Exam Objectives

Create and use ArrayList, TreeSet, TreeMap, and ArrayDeque objects.

Collections Overview

A *collection* is a generic term that refers to a container of objects.

The *Java Collections Framework* is a library of classes and interfaces in the `java.util` package that provides collections with different characteristics.

The most important interfaces are:

- `Collection`
  This is the base interface of the collection hierarchy and it contains methods like `add()` , `remove()` , `clear()` , and `size()` .
- `Iterable`
  Implementing this interface allows an object to be "iterable" with a for-each loop, through an `Iterator` , and with the new `forEach()` method.
- `List`
  Interface for collections which, one, store a group of elements that can be accessed using an index, and two, accept duplicates.
- `Set`
  Interface for collections which do not allow duplicate elements.
- `Queue`
  Interface for collections which store a group of elements in a particular order, commonly in a first-in, first-out order.
- `Map`
  Interface for collections whose elements are stored as key/value pairs.

Of the last four, `Map` is the only one that implements neither the `Collection` nor the `Iterable` interface, but still, it's considered a collection, because it contains a group elements.

List

List is the most common collection. You use when you want to allow (or do not care if there are) duplicate elements. You can even insert `null` elements (not all collections allow it).

Elements have an insertion order, but you can add elements at any position, since this position is based on a zero-based index, like an array.

In fact, the most used implementation, `ArrayList`, is actually implemented as an `Object` array under the hood.

The difference with using an array is that a `List` can grow automatically when elements are added. However, since it does that by copying the elements to a bigger array, adding (and removing) is slower.

Here are some basic `List` operations:

```
// Creating an ArrayList with an initial capacity of 10
List<String> list = new ArrayList<>(10);

System.out.println(list.isEmpty()); // true
list.add("a");
System.out.println(list.get(0)); // a
list.add(0, "b"); // Inserting b at index 0
list.add("c");
list.add(null);
System.out.println(list); // [b, a, c, null]
list.set(2, "a"); // Replacing element at index 2 with a
System.out.println(list.contains("d")); // false
// Returning the index of the first match, -1 if not found
System.out.println(list.indexOf("a")); // 1
// Returning the index of the last match, -1 if not found
System.out.println(list.lastIndexOf("a"); // 2

list.remove(1); // Removing by index
list.remove(null); // Removing null
list.remove("a") // Removing the first matching element

System.out.println(list.size()); // 1
```

Another popular implementation is `LinkedList`, a doubly-linked list that also implements the `Deque` interface (more about this interface later).

An easy way to create a `List` is using the `java.util.Arrays.asList` method:

```
String[] arr = {"a", "b", "c", "d"};
List<String> list = Arrays.asList(arr);
```

Or simply:

```
List<String> list =
        Arrays.asList("a", "b", "c", "d");
```

It returns an implementation of `List` backed by the specified array (but it's not an `ArrayList` and it doesn't implement all methods of `List`) that has fixed size, which means that you can't add elements to it. Also, modifications to the `List` are reflected in the original array.

Set

The main feature of a `Set` is that it doesn't allow duplicates.

The two most used implementations are `HashSet` and `TreeSet`. The difference between them is that `TreeSet` sorts the elements, while `HashSet` doesn't guarantee the order or that the order will remain constant over time.

`HashSet` stores the elements in a hash table (using a `HashMap` instance). Because of that, elements are not kept in order, but adding and looking up elements is fast.

To retrieve objects and avoid duplicates, the elements have to implement the `hashCode()` and `equals()` methods.

Here's an example of `HashSet`:

```
// Creating a HashSet with an initial capacity of 10
Set<String> set = new HashSet<>(10);
// add() returns true if the element is not already in the set
System.out.println(set.add("b")); // true
System.out.println(set.add("x")); // true
System.out.println(set.add("h")); // true
System.out.println(set.add("b")); // false
System.out.println(set.add(null)); // true
System.out.println(set.add(null)); // false
System.out.println(set); // [null, b, x, h]
```

As you can see, `HashSet` accepts `null` values.

`TreeSet` stores the elements in a red-black tree data structure. That's why it keeps the elements sorted and guarantees log(n) time cost for adding, removing, looking up an element, and getting the size of the set.

To avoid duplicates, the elements have to implement the `equals()` method. For sorting, elements have to either implement the `Comparable` interface (the implementation of `compareTo()` has to be consistent with the implementation of the `equals()` method) or pass an implementation of `Comparator` in the constructor. Otherwise, an exception will be thrown.

Here's an example similar to the previous one implemented with `TreeSet`:

```
Set<String> set = new TreeSet<>();
System.out.println(set.add("b")); // true
System.out.println(set.add("x")); // true
System.out.println(set.add("h")); // true
System.out.println(set.add("b")); // false
System.out.println(set); // [b, h, x]
```

Since `String` implements `Comparable`, and its `compareTo()` method implements lexicographic ordering, a `Set` is ordered that way.

Notice that this example doesn't add null values. That's because `TreeSet` doesn't accept them. If you try to add `null` to a `TreeSet`, a `NullPointerException` will be thrown.

This is because when an element is added, it's compared (as a `Comparable` or with a `Comparator`) against other values to insert it in the correct order, but it can't do that with a `null` value.

Queue

In a `Queue`, elements are typically added and removed in a FIFO (first-in-first-out) way.

The most used implementation is `ArrayDeque`, which is backed by an array, that has the functionality of adding and removing elements from both the front (as a stack) and back (as a queue) of the queue, and not in any position like in an `ArrayList`. This class doesn't allow inserting of `null` values.

Besides having the methods of `Collection`, `ArrayDeque` has other methods that are unique to queues. We can classify these methods into two groups:

Methods that throw an exception if something goes wrong:

- `boolean add(E e)`
  Adds an element to the end of the queue and returns true if successful or throws an exception otherwise.
- `E remove()`
  Removes and returns the first element of the queue or throws an exception if it's empty.
- `E element()`
  Returns the next element of the queue or throws an exception if it's empty.

Methods that return `null` if something goes wrong:

- `boolean offer(E e)`
  Adds an element to the end of the queue and returns `true` if successful or `false` otherwise.
- `E poll()`
  Removes and returns the first element of the queue or `null` if it's empty.
- `E peek()`
  Returns the next element of the queue or `null` if it's empty.

For each operation, there's a version that throws an exception and another that returns `false` or `null`. For example, when the queue is empty, the `remove()` method throws an exception, while the `poll()` method returns `null`.

```
Queue<String> queue = new ArrayDeque<>();
System.out.println(queue.offer("a")); // true [a]
System.out.println(queue.offer("b")); // true [a, b]
System.out.println(queue.peek()); // a [a, b]
System.out.println(queue.poll()); // a [b]
System.out.println(queue.peek()); // b [b]
System.out.println(queue.poll()); // b []
System.out.println(queue.peek()); // null
```

You can also use this class as a stack, a data structure that order the elements in a LIFO (last-in-first-out), when you use the following methods:

```
// Adds elements to the front of the queue
void push(E e)

// Removes and returns the next element
// or throws an exception if the queue is empty
E pop()
```

Notice that these methods are not in the `Queue` interface:

```
ArrayDeque<String> stack = new ArrayDeque<>();
stack.push("a"); // [a]
stack.push("b"); // [b, a]
System.out.println(stack.peek()); // b [b, a]
System.out.println(stack.pop()); // b [a]
System.out.println(stack.peek()); // a [a]
System.out.println(stack.pop()); // a []
System.out.println(stack.peek()); // null
```

Map

While a `List` uses an index for accessing its elements, a `Map` uses a key that can be of any type (usually a `String`) to obtain a value.

Therefore, a map cannot contain duplicate keys, and a key is associated with one value (which can be any object, even another map, or `null`).

The two most used implementations are `HashMap` and `TreeMap`. The difference between them is that `TreeMap` sorts the keys, but adds and retrieves keys in log(n) time while `HashMap` doesn't guarantee the order but adds and retrieves keys faster.

It is important that the objects used as keys have the methods `equals()` and `hashCode()` implemented.

Since `Map` doesn't implement `Collection`, its methods are different. Here's an example that shows the most important ones:

```
Map<String, Integer> map = new HashMap<>();

// Adding a key/value pair
System.out.println( map.put("oranges", 7) ); // null
System.out.println( map.put("apples", 5) ); // null
System.out.println( map.put("lemons", 2) ); // null
System.out.println( map.put("bananas", 7) ); // null

// Replacing the value of an existing key. Returns the old one
System.out.println( map.put("apples", 4) ); // 5
 System.out.println( map.size() ); // 4

// {oranges=7, bananas=7, apples=4, lemons=2}
System.out.println(map);

// Getting a value
System.out.println( map.get("oranges") ); // 7

// Testing if the map contains a key
System.out.println( map.containsKey("apples") ); // true
// Testing if the map contains a value
System.out.println( map.containsValue(5) ); // false

// Removing the key/value pair and returning the value
System.out.println( map.remove("lemons") ); // 2
// Returns null if it can't find the key
System.out.println( map.remove("lemons") ); // null

// Getting the keys as a Set
// (changes are reflected on the map and vice-versa)
Set<String> keys = map.keySet(); // [oranges, bananas, apples]

// Getting the values as a Collection
// (changes are reflected on the map and vice-versa)
Collection<Integer> values = map.values(); // [7, 7, 4]

// Removing all key/value pairs
map.clear();

System.out.println( map.isEmpty() ); // true
```

If we change the implementation to `TreeMap`, the map will be stored in a red-black tree structure and sorted just like a `TreeSet`, either by a `Comparator` or `Comparable`, with the natural order of its key by default:

```
Map<String, Integer> map = new TreeMap<>();

System.out.println( map.put("oranges", 7) ); // null
System.out.println( map.put("apples", 5) ); // null
```

```
System.out.println( map.put("lemons", 2) ); // null
System.out.println( map.put("bananas", 7) ); // null

// {apples=5 , bananas=7, lemons=2, oranges=7}
System.out.println(map);

// [apples, bananas, lemons, oranges]
Set<String> keys = map.keySet();
 Collection<Integer> values = map.values(); // [5, 7, 2, 7]
```

Notice that because of the way the sort is done (again, just like `TreeSet`); a `TreeMap` cannot have a `null` value as a key:

```
Map<String, Integer> map = new TreeMap<>();
map.put(null, 1); // throws NullPointerException!
```

However, a `HashMap` can:

```
Map<String, Integer> map = new HashMap<>();
map.put(null, 1); // OK
```

Key Points

- `Collection`
  This is the base interface of the collection hierarchy and it contains methods like `add()` , `remove()` , `clear()` , and `size()` .
- `Iterable`
  Implementing this interface allows an object to be "iterable" with a for-each loop, through an `Iterator` , and with the new `forEach()` method.
- `List`
  Interface for collections which, one, store a group of elements that can be accessed using an index, and two, accept duplicates.
- `Set`
  Interface for collections which do not allow duplicate elements.
- `Queue`
  Interface for collections which store a group of elements in a particular order, commonly in a first-in, first-out order.
- `Map`
  Interface for collections whose elements are stored as key/value pairs.

The following table compares the collections reviewed in this chapter:

Collection	**Interface**	**Implements Collection?**	**Allows duplicates?**	**Allows null values?**	**Ordered?**
`ArrayList`	`List`	Yes	Yes	Yes	Yes (Insertion Order)
`HashSet`	`List`	Yes	No	Yes	No
`TreeSet`	`List`	Yes	No	No	Yes (Natural order or by `Comparator`)
`ArrayDeque`	`Queue` `Deque`	Yes	Yes	No	Yes (FIFO or LIFO)
`HashMap`	`Map`	No	Just for values	Yes	No

`TreeMap`	`Map`	No	Just for values	No	Yes (Natural order or by `Comparator`)

Self Test

1. Given:

```
public class Question_7_1 {
    public static void main(String[] args) {
        ArrayDeque<Integer> deque =
              new ArrayDeque<Integer>();
        deque.push(1);
        deque.push(2);
        deque.push(3);
        deque.poll();
        System.out.println(deque);
    }
}
```

What is the result?
A. `[1, 2, 3]`
B. `[1, 2]`
C. `[2, 1]`
D. An exception occurs at runtime

2. Which of the following options can throw a `NullPointerException` ?
A.

```
TreeSet<String> s = new TreeSet<>();
s.add(null);
```

B.

```
HashMap<String> m = new HashMap<>();
m.put(null, null);
```

C.

```
ArrayList<String> arr = new ArrayList<>();
arr.add(null);
```

D.

```
HashSet<String> s = new HashSset<String>();
s.add(null);
```

3. Given:

```
public class Question_7_3 {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.remove(1);
        System.out.println(list);
    }
}
```

What is the result?
A. `[2, 3]`
B. `[1, 3]`
C. `[1, 2, 3]`
D. An exception occurs at runtime

4. Which of the following statements is true?
A. `HashSet` is an implementation of `Map`.
B. Objects used as values of a `TreeMap` are required to implement `Comparable`.
C. Objects used as values of a `TreeMap` are required to implement the `hashCode()` method.
D. Objects used as keys of a `TreeMap` are required to implement the `hashCode()` method.

Open answers page

---

06. Generics 08. Functional Interfaces